Прости алгоритми върху графи

Тодор Дуков

Защо изобщо се занимаваме с графи?

Графите са може би най-приложимата структура в областта на компютърните науки. Тяхната моделираща мощ е несравнима с тази на останалите структури. Те могат се използват за моделиране на:

- приятелски връзки в социални мрежи
- пътни мрежи в навигационни системи
- йерархични системи
- биологични мрежи

Някои от задачите, които могат да решават са:

- намиране на най-къс път от точка A до точка B
- намиране на съвместима наредба на дадени задачи
- намиране на най-добро разписание на полети
- класифициране на уебсайтове по популярност
- маркетинг в социални мрежи
- валидация на текст

Как представяме графите в паметта?

В зависимост от нашите цели графите могат да бъдат представени в паметта по различни начини. Най-използваните начини са:

- списък на съседство – за всеки връх се пазят в списък съседите му (и теглата ако има такива).
- матрица на съседство – пази се булева (може и числова ако графът е тегловен) таблица със всевъзможните комбинации от двойки върхове. Ако между два върха има ребро, то в съответната клетка пише единица (или теглото на реброто при тегловен граф), иначе нула.
- списък с ребрата – множеството от ребра идва като списък. Обикновено ако графът е неориентиран се пази само една пермутация на реброто.

Матрицата на съседство се използва по-рядко. Този подход е добър, когато графите са гъсти т.е. има много ребра в тях. В противен случай ние заемаме много повече памет от колкото ни е нужна. За сметка на това можем да проверим дали между два върха има ребро за константно време.
Списъците на съседство са по-пестеливи от към памет в средния случай, обаче за сметка на това по-бавно се проверява съседство между два върха. Този подход е добър, когато графите са редки т.е. имат малко ребра в тях. Също така ако по някаква причина ни трябва да изброяваме точно съседите на някакъв връх (да кажем за някакво обхождане), това очевидно е най-добрият начин. В най-лошия случай заемаме двойно повече памет от подхода с матрицата.
Списъка с ребрата е най-пестеливия начин от тези три. Пази се минималното количество нужна информация. Проблемът тук е, че проверката за съседство и изброяването на съседи на даден връх са бавни. Обаче това представяне все пак се използва, например когато искаме да построим МПД.
Накратко, сложностите са такива:

подход	памет	проверка за съседство	изброяване на съседите $N(v)$ на връх v
списък на съседство	$O(\|V\| + \|E\|)$	$O(\|V\|)$	$\Theta(\|N(v)\|)$
матрица на съседство	$\Theta(\|V\|^2)$	$\Theta(1)$	$O(\|V\|)$
списък с ребра	$\Theta(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$

Код на алгоритмите за обхождане на графи

```cpp
void dfs_helper(const vector<vector<int>> &g, int curr, vector<bool> &visited, vector<int> &result)
{
    result.push_back(curr);
    visited[curr] = true;

    for (const int &neighbour : g[curr])
        if (!visited[neighbour])
            dfs_helper(g, neighbour, visited, result);
}

vector<int> dfs(const vector<vector<int>> &g)
{
    int n = g.size();
    vector<bool> visited(n, false);
    vector<int> result;

    for (int i = 0; i < n; ++i)
        if (!visited[i])
            dfs_helper(g, i, visited, result);

    return result;
}

void bfs_helper(const vector<vector<int>> &g, int start, vector<bool> &visited, vector<int> &result)
{
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty())
    {
        int curr = q.front();
        result.push_back(curr);

        q.pop();

        for (const int &neighbour : g[curr])
            if (!visited[neighbour])
            {
                q.push(neighbour);
                visited[neighbour] = true;
            }
    }
}

vector<int> bfs(const vector<vector<int>> &g)
{
    int n = g.size();
    vector<bool> visited(n, false);
    vector<int> result;

    for (int i = 0; i < n; ++i)
        if (!visited[i])
            bfs_helper(g, i, visited, result);

    return result;
}
```

Задачи

Задача 1. Да се напише алгоритъм, който при подаден граф намира броя на свързаните компоненти.

Задача 2. Да се напише алгоритъм, който при подаден граф проверява дали той е цикличен.

Задача 3. Да се напише алгоритъм, който при подаден граф проверява дали той е дърво.

Задача 4. Да се напише алгоритъм, който при подаден граф (може и ориентиран) и два негови върха намира най-късия път между тях.

Задача 5. Да се напише алгоритъм, който при подаден граф (може и ориентиран), два негови върха и дължина намира броят на пътища между тях със съответната дължина.

Задача 6. Да се напише алгоритъм, който при подаден граф проверява дали той е двуделен.

Задача 7. Трябва да изпълним задачи $1, \ldots, n$. Обаче имаме допълнителни изисквания $R[1 \ldots k]$ от вида $\langle i, j \rangle$, които казват *"задача i трябва да се изпълни преди задача j"*. Да се напише алгоритъм, който при подадени изисквания намира последователност от задачи, която удовлетворява тези изисквания. Ако няма такива, да се върне съобщение за грешка.

Задача 8. Да се напише алгоритъм, който при подаден ориентиран тегловен граф намира теглата на минималните пътища между всеки два върха.